Date of acceptance Grade

Instructor

Performance Evaluation of Bloom Multifilters

Francesco Concas

Helsinki December 7, 2017

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Francesco Concas

Työn nimi — Arbetets titel — Title

Performance Evaluation of Bloom Multifilters

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	December 7, 2017	57 pages + 1 appendices

Tiivistelmä — Referat — Abstract

The Bloom Filter is a space-efficient probabilistic data structure that deals with the problem of set membership. The space reduction comes at the expense of introducing a false positive rate that many applications can tolerate since they require approximate answers. In this thesis, we extend the Bloom Filter to deal with the problem of matching multiple labels to a set, introducing two new data structures: the Bloom Vector and the Bloom Matrix. We also introduce a more efficient variation for each of them, namely the Optimised Bloom Vector and the Sparse Bloom Matrix. We implement them and show experimental results from testing with artificial datasets and a real dataset.

ACM Computing Classification System (CCS):
Mathematics of computing∼Probabilistic algorithms
Information systems∼Data compression

Avainsanat — Nyckelord — Keywords
Bloom filters, Bloom multifilters, Bloomier filter, false positive rate

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Fast matching of arbitrary identifiers to certain values is the basic requirement of applications in which data objects are typically referred to by using local or global unique identifiers, usually called labels [KT06]. In a typical usage scenario, each label maps to a small set of values [KT13]. For example, in order to provide lower latency for accessing data, such data is cached across different regions; given a set of such contents, we need to find mapping cached contents to the proxy servers.

Many popular Internet services, including Google Search, Yahoo Directory, and web-based storage services [BP12], rely on solutions for efficient data matching, many of which rely on custom techniques for providing scalable, fault-tolerant, and low-cost services [PBC00, Rab89, SKH95]. A popular probabilistic data structure called Bloom Filter serves a similar purpose, where it answers whether a label belongs to a particular set or not.

The Bloom Filter is a data structure which represents a set of labels using a fixed amount of bits, which is very space efficient, with the downside that it can yield false positives to queries. The Bloom Filter, however, works only for a single set. There are many extensions of the Bloom Filter, many of which represent multiple labels in a set, or that can answer whether a label is present or not in any of multiple sets.

In this thesis, we introduce probabilistic data structures based on the Bloom Filter which represent multiple sets, that given a label as a query, yield as a response which sets contain that label. These are our contributions:

- We introduce two new Bloom Multifilters, the Bloom Matrix and the Bloom Vector. These data structures are inspired by the standard Bloom Filter and some of its extensions [ABPH07, Blo70, BMP$^+$06, GWC$^+$10, Mit02, Mul83], in particular, the Bloomier Filter [CKRT04] and the Bloom Multifilter [XLR16].
- We provide a theoretical analysis of our data structures, for time and space complexity and for false positive rate.
- We show a Scala implementation and we test them with two randomly generated datasets, each following a different distribution: namely uniform and Zipf.

This thesis is organised into three parts. In the first part, we introduce the theoretical background needed to understand our work, the Bloom Filter, and we describe

the problem that we tackle with our Bloom Multifilters. In the second part, we introduce our Bloom Multifilters and we show a Scala implementation. In the third and last part, we show the results of experiments on randomly generated datasets and conclude our thesis.

Part I

Preliminaries

In the first part of this thesis, we introduce some theoretical concepts which are necessary to understand our work. We will discuss hash functions, error rates, Bloom Filter, and some of its extensions that inspired our work. We also introduce the problem that we later show how to deal with by using our Bloom Multifilters.

2 Theoretical Background

In this section, we introduce hash functions and error rates. Hash functions are an essential component of Bloom Filters, and the concept of error rates is necessary to evaluate the performance of probabilistic data structures such as the Bloom Filter.

2.1 Hash functions

A hash function is a mathematical function which maps a given input of arbitrary size, usually a string, into a value of fixed size. The result is called *hash value*, or simply *hash*. Hash functions are useful in cryptography, caches, string searching, and the applications in which we are interested in: hash-based lookup applications. There are different types of hash functions, each one optimised for its purpose. Usually, the result of a hash function is a 32 or 64-bit integer, but in cryptographic applications, it can be as large as 256 or 512 bits.

The fixed size of the hash implies that the output of a hash function is restricted to a defined range, therefore there can be some occurrences in which different input values can map to the same output value. These occurrences are called *collisions*.

2.1.1 Properties of hash functions

In many applications hash functions must satisfy some properties. Here we explain some of them that must be satisfied by the hash functions that we use in our work.

Determinism For a given input value, the hash function must always return the same hash value. This applies especially to applications in which we compare data

using hashes. For example, let us suppose that we just transferred a big file over the network and we need to check whether the transfer was successful. We can compute the hash of the two files on each machine and compare them; if the hashes are equal it means that there is a very high probability that the transfer was successful. Although such result could be a collision, with a large enough range there is a very low probability of it happening.

Uniformity Given several input values, their resulting hash values must follow a uniform distribution, which means that different hash values must have the same probability of being generated. This is useful to reduce the probability of collisions to a minimum.

Defined range The values returned by a hash function must belong to a predefined range. Usually, the range of a hash function is $[0, 2^n - 1]$, with n usually equal to 32 or 64. We can restrict its range by taking, as a result, the remainder of the division of the hash by a number m, which restricts its range to $[0, m - 1]$. This is the technique that we use in our work.

2.1.2 MurmurHash

MurmurHash is a hash function used in hash-based lookup applications [YN13, YCL16]. It takes two parameters, an input and a seed. The output is computed on the input and it is based on the seed. Therefore, ideally, for different seeds, the hash function produces different outputs on the same input.

We chose Murmurhash for the implementation of our Bloom Multifilters for two reasons. Firstly, because it can generate different outputs with the same input, using different seeds. This allows us to easily emulate the use of different hash functions, which is what we need. Secondly, we used Scala to implement our Bloom Multifilters, which has an implementation of MurmurHash in its API.

2.2 Error rates

Let us introduce the concept of error rates with an example related to our work. Let us assume that we have a set of elements. We wish to classify whether each of them belongs or not to another set. Let us also assume that we have a probabilistic algorithm that classifies whether an element belongs or not to such set, giving the

correct answer with a certain probability p. We can have four different types of outcomes, which we can store in what is called a *confusion matrix*:

		Predicted value: True	Predicted value: False
Actual value	True	True Positive (TP)	False Negative (FN)
	False	False Positive (FP)	True Negative (TN)

There are several statistical measures to measure the performance of a classifier. In this thesis, we are interested in only one among such measures, namely the false positive rate, or FPR:

$$FPR = \frac{FP}{FP + TN} \tag{1}$$

3 The Bloom Filter

The Bloom Filter is a probabilistic data structure proposed by Burton H. Bloom [Blo70], the purpose of which is to represent a set so that it will occupy much less memory space than it normally would when represented with conventional methods. This comes at the cost of introducing an FP rate. FNs, on the other hand, are not allowed.

A query on a Bloom Filter, which checks whether an element is or not in the represented set, can return either a positive or a negative response. If positive, the result means that the element is probably in the set. If negative, it means that the element is definitely not in the set, because as we stated in the previous paragraph, FNs are not allowed.

The original application that Bloom proposed was to reduce the number of unnecessary disk accesses to retrieve data [Blo70]. The idea behind this is that if the Bloom Filter produces a negative result from an input, there is no need to access the disk. If the FP rate is low enough, only a small portion of the inputs that produce a positive will make the system to access the disk unnecessarily.

Bloom Filters nowadays are used in many applications, particularly in networks [BM04]. They are also used for differential file access [Gre82] and joins and semi-joins on distributed queries [LR95, ML86, Mul90]. Bloom filters are used also in applications involving set representations, including Akamai [MS15], Foundation [RCP08], SPIN [Hol03], and Google BigTable [CDG+08].

3.1 Definition

Let us formally define the Bloom Filter. It can be seen as a probabilistic function that returns true whether an element probably belongs to the represented set or false if it definitely does not belong to such set.

Let U be a set of all the items that we can possibly store, and let $E \subseteq U$ be the set that we wish to represent. We are interested in encoding the function $f : U \to \{0, 1\}$ defined as:

$$f(x) = \begin{cases} 1 & \text{if } x \in E \\ 0 & \text{otherwise} \end{cases}$$

The Bloom Filter is defined as a pair (B, H), where B is a bitset of size m and $H = \{h_1(x), ..., h_k(x)\}$ is a set of hash functions, each having image $[0, m-1]$. We also define two operations on Bloom Filter: $\textsc{Add}(x)$ and $\textsc{Lookup}(x)$. We do not define a remove operation because it would introduce a chance of FNs. We will explain later in this subsection why.

The set of distinct values returned by all the hash functions, given in input an element x, is called its *neighbourhood*; we define it, with abuse of notation, as $H(x)$.

Add When a Bloom Filter is created, the bits in its B are initialised to 0. Whenever we add an element x, we set to 1 each $B[i]$ for each $i \in H(x)$:

$$F.\,\textsc{Add}(x) := B[h_i(l)] \leftarrow 1 \text{ for } 1 \leq i \leq k \tag{2}$$

Lookup To test the membership of an element x, we have to check whether all of the bits $B[i]$ for each $i \in H(x)$ are set to 1. If it is true, then the element is probably in the set, otherwise, it is definitely not in the set:

$$F.\,\textsc{Lookup}(x) := \bigwedge_{1 \leq i \leq k} B[h_i(l)] \tag{3}$$

Multiple Lookup To test the membership of multiple elements, more formally a set of elements X, we compute the hash neighbourhood of all the elements, and use the function defined in Equation 3.

Theorem 1. *A Bloom Filter has a FN rate always equal to 0.*

Proof. Let us assume that a Bloom Filter returns a false value for an element x previously added to it. This would imply that at least one of the bits to which it is

mapped to is set to 0. Since all of its bits were set to 1 during the add operation, and there are no operations that would set them back to 0, this is a contradiction. □

Following this, we can now deduce why we cannot define a remove operation on a Bloom Filter: setting a bit in which there is a collision to 0 would introduce a chance of FNs.

3.2 False positive rate

False positives occur whenever we look for an element x which is not in the set and the LOOKUP function returns true. Such function returns true whenever all the bits having as indexes the neighbourhood of x are set to 1; this implies that the more bits are set to 1, the higher is the false positive rate. Since the number of bits set to 1 increases by adding elements, the false positive rate increases with the number of elements inserted.

The false positive rate is influenced also by the number of hash functions. A higher number of hash functions decreases the chance of collisions between two different elements. However, since it also increases the number of bits set to 1, the optimal number of hash functions is a compromise.

Bose et al. [BGK+08] have shown that the probability p of false positives in a Bloom Filter of size m and with k hash functions is:

$$p = \Theta\left(\left[1 - \left(1 - \frac{1}{m}\right)^{kn}\right]^k\right) \tag{4}$$

$$= \Theta\left(\left(1 - e^{-kn/m}\right)^k\right) \tag{5}$$

If we know a priori the number of elements that we are going to insert in a Bloom Filter, we can choose its parameters so that the Bloom Filter will have a probability of false positives around a certain value p. We derive from Equation 4:

$$m = -n\frac{\ln p}{\ln^2 2} \tag{6a}$$

$$k = \ln 2 \cdot \frac{m}{n} = -\log_2 p \tag{6b}$$

Example 3.1. Let us assume that we wish to create a Bloom Filter with a false positive rate of approximately 1%. According to equation 6a, we need for each

element a number of bits equal to

$$\frac{m}{n} = -\frac{\ln 0.01}{\ln^2 2} \approx 10$$

This is much less than the number of bits required to represent an element such as a string when using conventional data structures.

3.3 Cardinality estimation

Let X be the number of bits set to 1 in a Bloom Filter with size m and having k hash functions. It is possible to compute the approximate number of elements n^* in such Bloom Filter [SB07], by using the equation:

$$n^* = -\frac{m}{k} \ln \left[1 - \frac{X}{m} \right] \tag{7}$$

3.4 Set operations

Since a Bloom Filter is an approximate representation of a set, we can perform some set operations between two Bloom Filters. In order to do so, they must be compatible, meaning that they must have the same size and hash functions.

Union The union operation between two Bloom Filters is performed by computing the OR operation on their bits having the same indexes. Let F_1 and F_2 be two Bloom filters, B_1 and B_2 their respective bitsets having equal size m, and H_1 and H_2 their respective hash functions, with $H_1 = H_2$. We can define the union as:

$$F_1 \cup F_2 := (B_\cup, H_\cup) \tag{8}$$

where $H_\cup = H_1 = H_2$, and $B_\cup$ is a bitset of size m defined as:

$$B_\cup[i] = B_1[i] \vee B_2[i] \text{ for } 1 \leq i \leq m$$

Intersection Similarly, the intersection operation on two Bloom Filters is performed by computing the AND operation on their bits having the same indexes. Using the same definitions, we can define the intersection as:

$$F_1 \cap F_2 := (B_\cap, H_\cap) \tag{9}$$

where $H_\cap = H_1 = H_2$, and $B_\cap$ is a bitset of size m defined as:

$$B_\cap[i] = B_1[i] \wedge B_2[i] \text{ for } 1 \leq i \leq m$$

We cannot implement the other set operations, namely complement and difference, because we would have to implicitly remove elements from one of the Bloom Filters. We already discussed that it is not allowed because it would introduce a probability of FN.

3.5 An example implementation

Let us now see an example implementation of a Bloom Filter. We need a bitset B of size m, and k different seeds for MurmurHash, to simulate the use of k different hash functions. We could also simply use the range $[1, k]$ so that we need only an integer to represent k. Of course, the results of the hash functions should also be restricted to the range $[1, k]$ by computing the remainder of the division by k on them. The space occupied by a Bloom Filter is clearly $\Theta(m)$.

Add To implement the ADD operation, we create a function that takes a string as a parameter and computes k different hashes using the k different seeds, setting to 1 all bits having as indexes the results (Algorithm 1).

Algorithm 1 Bloom Filter ADD operation

```
1: procedure ADD(l)
2:     for i ← [1, k] do
3:         B[MURMURHASH(l, i) mod k] ← 1
4:     end for
5: end procedure
```

Lookup To implement the LOOKUP operation, we create a function that works almost exactly the same way, with the exception that instead of setting bits, it checks whether they are all set to 1 (Algorithm 2).

Algorithm 2 Bloom Filter LOOKUP operation

```
1: procedure LOOKUP(l)
2:     for i ← [1, k] do
3:         if B[MURMURHASH(l, i) mod k] = 0 then
4:             return 0
5:         end if
6:     end for
7:     return 1
8: end procedure
```

Multiple Lookup We can use Algorithm 2 to create a function for looking up multiple labels at once. We can see a pseudocode in Algorithm 3.

Algorithm 3 Bloom Filter multiple LOOKUP operation

```
1: procedure LOOKUP(L)
2:     for each l ∈ L do
3:         if LOOKUP(l) = 0 then
4:             return 0
5:         end if
6:     end for
7:     return 1
8: end procedure
```

We can clearly see that the time complexity of ADD is $\Theta(k)$, and for LOOKUP is $O(k)$, $O(|L|k)$ in the multiple label case. Let us now show the set operations.

Union As we previously discussed, we need to perform the OR operation on the bits having same indexes in **B**. We can see an implementation in Algorithm 4.

Algorithm 4 Bloom Filter union operation

```
1: procedure UNION((B_1, k_1), (B_2, k_2))
2:     if |B_1| = |B_2| ∧ k_1 = k_2 then
3:         Let B_∪ be a new bitset of size |B_1|
4:         for i ← [1, k] do
5:             B_∪[i] ← B_1[i] ∨ B_2[i]
6:         end for
7:         return (B_∪, k_1)
8:     end if
9: end procedure
```

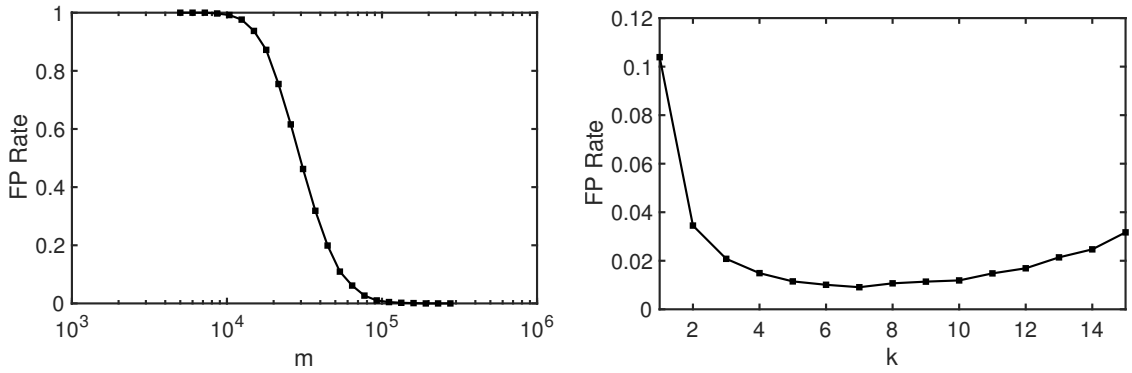
Intersection Similarly, we can implement the intersection by using the AND operator in line 4, as we see in Algorithm 5.

Algorithm 5 Bloom Filter intersection operation

```
1: procedure INTERSECTION((B_1, k_1), (B_2, k_2))
2:     if |B_1| = |B_2| ∧ k_1 = k_2 then
3:         Let B_∩ be a new bitset of size |B_1|
4:         for i ← [1, k] do
5:             B_∩[i] ← B_1[i] ∧ B_2[i]
6:         end for
7:         return (B_∩, k_1)
8:     end if
9: end procedure
```

3.6 Testing the Bloom Filter

Now that we defined an implementation for the Bloom Filter, we can test it by using randomly generated labels. In our case, we generated 20000 labels and randomly selected half of them to be added to the Bloom Filter. We used the other half as the test dataset. We tested the Bloom Filter with different m and k, the results are shown in Figure 1.



(a) Variable m with $k = 7$.

(b) Variable k with $m = 96000$

Figure 1: Bloom Filter FP rate by different values of m and k, representing a randomly generated set of 10000 items.

4 Related work

There are many extensions of the Bloom Filter. In this section, we discuss some of them.

4.1 Counting Bloom Filter

The Counting Bloom Filter [FCAB00] is a variant of the Bloom Filter that allows a delete operation without creating the chance of FNs. It works almost exactly as the standard Bloom Filter, but instead of using a bitset, it uses an array of integers.

The add operation increments the integers to which an element is mapped to with the hash functions, while the delete operation decrements them. The lookup function simply checks whether all of the integers to which an element are mapped to are higher than 0.

A downside of the Counting Bloom Filter is that it occupies much more memory than a standard Bloom Filter since it uses integers instead of bits. Another downside is that it is also vulnerable to arithmetic overflow.

4.2 Compressed Bloom Filter

The Compressed Bloom Filter [Mit02] was proposed to reduce the number of bit broadcast in network applications, FP rate, and lookup time. This advantage comes at the cost of introducing a processing time for compression and decompression. The

compression algorithm proposed in the original work [Mit02] is Arithmetic Coding [MNW98], which is a lossless data compression technique.

As we have seen in Section 3, in a Bloom Filter we choose k for obtaining a certain FP rate. This is assuming that also an optimal m has been chosen, effectively basing the choice of k on m and the number of elements to be inserted. In a Compressed Bloom Filter, however, the optimal k is instead chosen to optimise the result of the compression algorithm, or the size of the resulting Bloom Filter after its compression. This results in a choice of k lower than in a standard Bloom Filter.

4.3 Split Bloom Filter

The Split Bloom Filter [CcFL04] uses a bitset split in multiple bins. Each bin has an associated hash function, and the hash functions are all different from each other. Whenever an element is added, it is added to all bins.

More formally, a Split Bloom Filter is composed by k bins $G = \{B_1, ..., B_k\}$ each having size m, where k is also the number of hash functions. Each hash function $h_i(x)$ is associated to the bitset B_i having the same index. Whenever an element x is added, we set to 1 the bits $B_i[h_i(x)]$, for $1 \leq i \leq k$.

The lookup operation works similarly, but it checks whether all bits $B_i[h_i(x)]$, for $1 \leq i \leq k$, are set to 1.

4.4 Scalable Bloom Filter

The Scalable Bloom Filter [ABPH07] is an improvement of the Bloom Filter which is useful in situations where the number of elements is not known, and an upper bound on the FP rate is still required.

A Scalable Bloom Filter starts with a Split Bloom Filter with k_0 bins and P_0 expected FP rate, which can support at most a number of elements that keep the FP rate

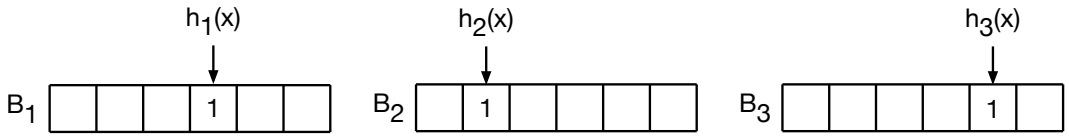


Figure 2: Example of a Split Bloom Filter with $k = 3$ and $m = 6$.

below P_0. When the filter gets full, another one is added with k_1 bins and $P_1 = P_0 r$ expected FP rate, where r is a tightening ratio decided during the implementation.

This leads to an upper bound on the total FP rate, which is:

$$P \leq P_0 \frac{1}{1-r} \tag{10}$$

The number of bins for each filter is:

$$k_0 = \log_2 P_0^{-1} \qquad k_i = \log_2 P_i^{-1} = k_0 + i \log_2 r^{-1} \tag{11}$$

Almeida et al. [ABPH07] found through experimentation that the optimal r is around 0.8–0.9.

The most interesting part of the Scalable Bloom Filter is the flexible growth. An initial bin size m_0 is chosen for the first Bloom Filter, then for the ith filter the size of each bin is:

$$m_i = m_0 s^i \tag{12}$$

where s is the growth factor. Almeida et al. [ABPH07] found that in practice an $s = 2$ is useful because if m_0 is a power of 2 each m_i will be also a power of 2. This is good because the range of a hash function is usually a power of 2.

4.5 Bloomier Filter

As we have seen in Section 3, The Bloom Filter can encode only Boolean functions. The Bloomier Filter [CKRT04] was proposed to represent arbitrary functions on finite sets.

Let $E = \{e_1, ..., e_N\}$ and $R = \{1, ..., |R| - 1\}$. Let $A = \{(e_1, v_1), ..., (e_N, v_N)\}$ be an assignment, where $v_i \in R$ for $1 \leq i \leq N$. We are interested in encoding such assignment, which can also be seen as a function $f : E \to R$ defined as:

$$f(x) = \begin{cases} v_i & \text{if } x \in E \\ \varnothing & \text{otherwise} \end{cases}$$

The Bloomier filter uses a bit matrix to encode the function previously defined. In order to build such a matrix, it uses a non-trivial algorithm, which can be found

in the original work [CKRT04]. In this algorithm, the Bloomier Filter uses two functions called ENCODE and DECODE. In Section 5 we define two similar functions that we call with the same names, which our Bloom Multifilters use.

4.6 Bloom Multifilter

The closest work related to ours is the Bloom Multifilter, which was devised by Xu, Liu and Rao [XLR16] to extend the Bloom Filter for solving the problem of multiple elements check on multiple sets at once.

Let $\mathcal{S} = \{S_1, ..., S_n\}$, where each S_i is a set of multiple elements. We are interested to check whether there is an $S \in \mathcal{S}$ which contains all the elements in a query q. More formally, we are interested in implementing a Boolean function f defined as:

$$f(q) = \begin{cases} 1 & \exists S \in \mathcal{S} : q \subseteq S \\ 0 & \text{otherwise} \end{cases}$$

Similarly to the Bloomier Filter, the Bloom Multifilter uses a bit matrix to represent multiple sets, and each set has an assigned ID. Whenever an element is added to such set, it is mapped to the rows having the indexes equal to its hash neighbourhood; the ID of the set, represented in binary, is then added to such rows using the bit-wise OR operation.

To check whether multiple elements belong to one of the sets, they are mapped to multiple rows according to their hash neighbourhood, then the bit-wise AND operation is performed on such rows. If the result is a value greater than 0, it means that those values are probably all contained in one of the sets. The Bloom Multifilter, however, does not answer in which sets all the elements are contained.

Xu, Liu and Rao [XLR16] also describe some improvements to this data structure, such as partitioning the sets into multiple Bloom Multifilters to balance the term frequency, in case that the distribution of the elements is skewed.

5 Problem definition and possible applications

In this section, we define the problem that we later show how to deal with Bloom Multifilters that we introduce in this study. We also describe some possible practical applications.

5.1 Definition

Both standard Bloom Filter and Bloomier Filter were meant to encode only one set, and the Bloom Multifilter can only answer whether it is true or false that one or multiple elements are contained in one of the represented sets. In this thesis, we extend the standard Bloom Filter to encode multiple sets and to efficiently check the membership of an element in all the sets. Furthermore, we extend it to answer to which of the sets such element belongs to.

Let $L = \{l_1, ..., l_{|L|}\}$ be a set of labels and $E = \{e_1, ..., e_N\}$ be a set of items. We are interested in representing the function $f : L \to \mathcal{P}(E)$, where $\mathcal{P}(E)$ is the power set of E.

The simplest approach is to use a Bloom Filter associated with each item, in which we can store the labels associated with such item. We also show another approach that has both advantages and disadvantages compared with the former, depending on the type of distribution of the data. The idea of the latter approach is to represent the function f similarly to a Bloom Filter. However, in this case, instead of using single bits to encode an element, we use bitsets in which we store binary representations of the element $s \in \mathcal{P}(E)$ to which labels maps to. We obtain these representations with two functions that we now introduce: ENCODE and DECODE.

5.2 Encode and Decode

Let Π be a total ordering on E. We introduce two functions: $\textsc{Encode}(\Pi, S)$, which given an ordering Π of E and a set of items $S \in \mathcal{P}(E)$ returns a binary representation $V = \{v_1, ..., v_N\}$ of S, such that:

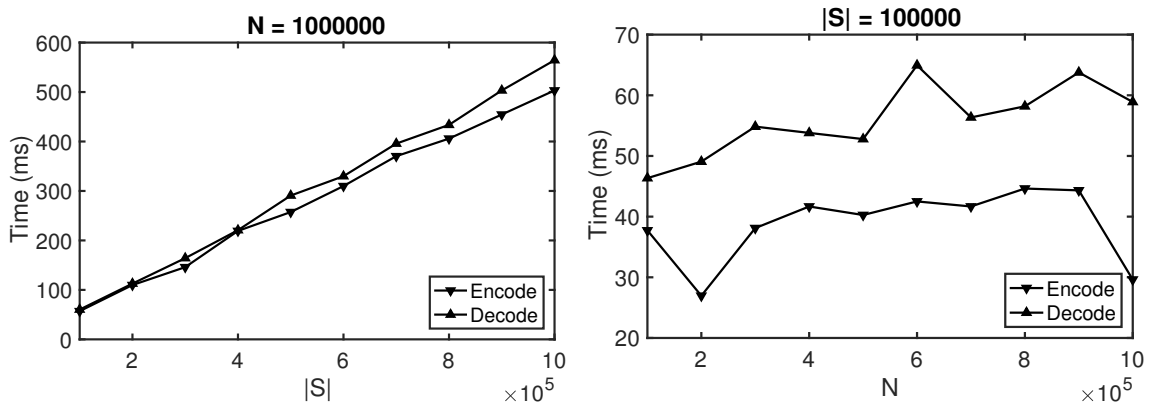
$$v_i = \begin{cases} 1 & \text{if } \Pi(i) \in S \\ 0 & \text{otherwise} \end{cases}$$

and $\textsc{Decode}(\Pi, V)$, which given an ordering Π of E and a binary representation V of a set of items $S \in \mathcal{P}(E)$ returns S.

Example 5.1. Let $E = \{e_1, e_2, e_3\}$ with Π following the same order. We have:

$$\textsc{Encode}(\Pi, \{e_1, e_3\}) = 101$$

$$\textsc{Decode}(\Pi, 011) = \{e_2, e_3\}$$



(a) Running time by $|S|$ (b) Running time by N

Figure 3: ENCODE and DECODE running time.

5.2.1 Encode and Decode implementation

To implement the ENCODE and DECODE functions, we need to represent the ordering Π in some way. We can use an array of strings, in which the operation to get the label of an item given its index in the ordering is constant in time. If we wish to do the inverse, namely to get its index given its label, in the worst case we need to scan the whole array. We can speed up the last operation by using also a hashmap Π_m, which reduces the time complexity to near constant (Algorithms 6, 7).

Algorithm 6 ENCODE operation

```
1: procedure ENCODE(Π_m,S)
2:     let V be a new bitset
3:     for each l ∈ S do
4:         V[Π_m[l]] ← 1
5:     end for
6:     return V
7: end procedure
```

5.2.2 Analysis of Encode and Decode

The ENCODE(Π_m, S) function needs to initialise a bitset V to 0 and to set some bits to 1, in order to return the encoded value of S. The space complexity is, therefore, $\Theta(N)$. For each element in S, we need to set $V[i] \leftarrow 1$, where i is the index of S in the ordering Π. Since we are using Π_m, the time complexity is $O(|S|)$.

Algorithm 7 DECODE operation

```
1: procedure DECODE(Π,V)
2:     let L be a new list
3:     for each i ∈ V do
4:         L.ADD(i)
5:     end for
6:     return L
7: end procedure
```

The $\textsc{Decode}(\Pi, V)$ function needs to create a set from the bitset V, which can be at most N, therefore it takes $O(N)$ space. For each bit v_i set to 1 in V, the function fetches the element at position i in the array and adds it to the set. If we need to return an ordered set, it takes $O(|V| \log |V|)$ time, otherwise it takes $O(|V|)$ time. In the rest of this thesis, we assume that we do not need to return an ordered set.

In Figure 3 we can see that $|S|$ influences the running time of ENCODE and DECODE much more than N.

5.3 Possible applications

In this subsection, we discuss some applications in which Bloom multifilters could be useful: load balancing and cache servers.

5.3.1 Load balancing

Load balancing is a technique used to optimally distribute workload between multiple computing resources, in order to maximise the performance of the whole system [Rab89, Cyb89, BW89, SKH95]. For example, a system can provide the same service simultaneously on multiple servers, in order to increase the total amount of bandwidth available. Load balancing is also useful for redundancy purposes: if there are multiple servers that provide the same service and one fails, the service remains online.

Let us assume that we have a cluster of servers which provides multiple services, with the services distributed across different servers. Reasoning as in the problem previously defined, the services can be represented as the labels L and the servers can be represented as the elements E. One of our Bloom multifilters could then be

used to map the services to the servers which provide them.

5.3.2 Web cache

A web cache temporally stores Internet content locally, so that it can be accessed without accessing the provider again. This reduces bandwidth usage and loading time. There are two types of web cache: browser cache, which is a cache integrated into all modern browsers, and proxy cache, which is an intermediate server between the client and the provider.

Much of the modern Internet is composed of proxy servers. This allows the workload to be split, providing also redundancy and an extra level of security due to the provider being hidden by the proxy servers. Whenever a client wishes to access a service from the provider, the request from the client is handled by one of the proxy caches instead. If the data is available and up-to-date, the proxy cache sends it immediately to the client. If not, the proxy cache fetches it from the provider and provides it to the client, also updating the local data to make it available for future requests.

In a proxy cache, Internet contents tend to be small compared to the number of requests. If we represent the requests as the labels L and the contents as the elements E, we could apply our Bloom multifilters to a proxy cache.

Part II

Bloom Multifilters

In this part, we discuss the Bloom Multifilters to solve the problem that we presented in Section 5.

The Bloom Vector is a vector of Bloom Filters, in which they can have different size and hash functions from each other. We will see later that this makes the Bloom Vector good for representing sparsely distributed data with an optimal amount of space.

The Bloom Matrix is a Boolean matrix, which works similarly to the Bloom Vector but in a more complex fashion. We will see later that the Bloom Matrix is useful for representing uniformly distributed data, and it is much faster than the Bloom Vector.

We also discuss concurrent operations and their implementations on our Bloom Multifilters.

6 Bloom Vector

Let us introduce the simplest approach, namely the Bloom Vector. It consists of a vector of Bloom Filters, in which each Bloom Filter represents the set of labels associated with a certain item of the data.

6.1 Definition

We define a Bloom Vector as a triplet (G, Π, Π_m), where G is an array of size $N = |E|$ in which each item corresponds to a Bloom Filter, and Π and Π_m represent an ordering on the set E as previously defined. As already stated, the Bloom Filters in G can have different sizes and different hash functions from each other. A Bloom Vector could be seen as a sparse binary matrix.

6.2 Operations

Let us now define the operations on the Bloom Vector, which are based on the operations on Bloom Filter.

6.2.1 Add

Whenever we wish to add a label l to the subset of $\mathcal{P}(E)$ given by $f(l)$, we compute $\textsc{Encode}(\Pi, f(l))$. Let us call I the indexes of the bits set to 1 in $\textsc{Encode}(\Pi, f(l))$. The add function in the Bloom vector F, is defined as:

$$F.\,\textsc{Add}(l) := G[i].\,\textsc{Add}(l) \quad \forall i \in I$$

We can see a pseudocode of the Add operation in Algorithm 8.

Algorithm 8 BV Add

```
1: procedure ADD(l, e)
2:     V ← ENCODE(Π_m, e)
3:     I ← V.toList
4:     for i ← I do
5:         G[i].ADD(l)
6:     end for
7: end procedure
```

6.2.2 Lookup

Similarly, whenever we wish to find out which subset of $\mathcal{P}(E)$ is labelled with l, let V be a bitset defined, for $1 \leq i \leq N$ as:

$$V[i] := \begin{cases} 1 & \text{if } G[i].\,\textsc{Lookup}(l) \\ 0 & \text{otherwise} \end{cases}$$

The Lookup operation is defined as:

$$F.\,\textsc{Lookup}(l) := \textsc{Decode}(\Pi, V)$$

Let us also show a pseudocode implementation in Algorithm 9.

Algorithm 9 BV Lookup

```
1: procedure LOOKUP(l, e)
2:     Let V be an empty bitset
3:     for i ← [0, N) do
4:         if G[i].LOOKUP(l) then
5:             V[i] ← 1
6:         end if
7:     end for
8:     return DECODE(Π, V)
9: end procedure
```

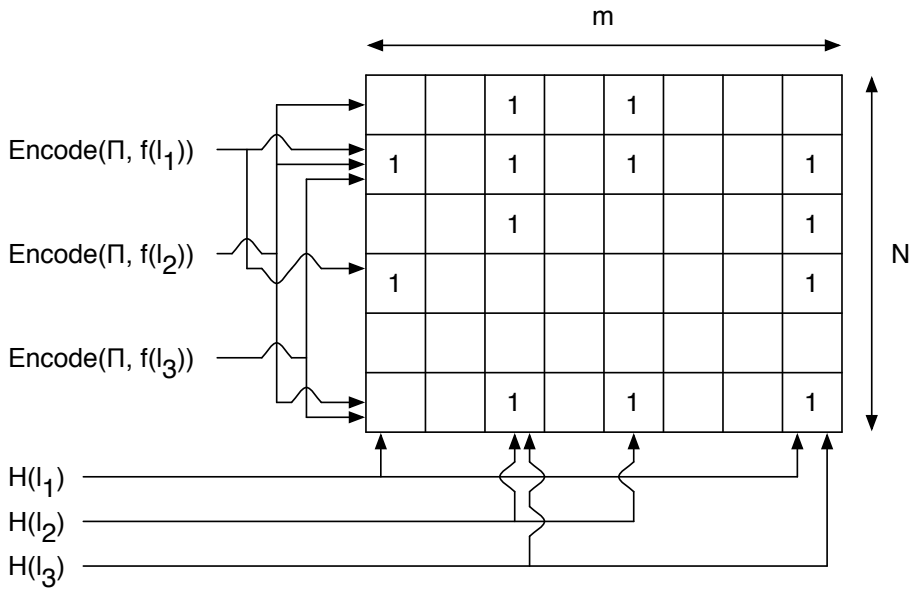


Figure 4: Example of a Bloom vector. The 1 in cell $\mathbf{G}[5, 2]$ is a collision.

Example 6.1. Let us assume that we have $E = \{e_1, ..., e_6\}$ with an ordering Π that follows the same order and $L = \{l_1, l_2, l_3\}$, and that we wish to represent a function $f : L \to \mathcal{P}(E)$ such that $f(l_1) = \{e_2, e_4\}$, $f(l_2) = \{e_1, e_2, e_6\}$ and $f(l_3) = \{e_3, e_6\}$. Let us also assume that we create a Bloom vector with all its Bloom filters having $m = 8$ and $H = \{h_1(x), h_2(x)\}$ such that $H(l_1) = \{0, 7\}$, $H(l_2) = \{2, 4\}$ and $H(l_3) = \{2, 7\}$. We obtain a configuration as shown in Figure 4.

We can easily notice that LOOKUP(l_1) and LOOKUP(l_2) both yield correct results, but LOOKUP(l_3), which performs the bitwise AND operation on columns $\mathbf{G}[\_, 2]$ and $\mathbf{G}[\_, 7]$, returns $\{e_2, e_3, e_5\}$, which contains a FP (e_2).

Multiple label lookup We can look up which items contain multiple labels by using the same algorithm for a single lookup, using in each Bloom Filter, the function for looking up multiple labels.

6.2.3 Loading data from a file

We also define another operation to load the data directly from a file. The file should be CSV formatted so that each line represents an item and its assigned labels. The first value of a line should be the name of the item, and the rest of the values should be the names of its assigned labels. The function simply processes the file line by line, then it prepends the item of each line to Π and the Bloom Filter representing its corresponding items to G.

We do not show a pseudocode of the load operation here, we will show the actual code implementation in Section 8 instead.

6.3 False positive rate

Let us recall Equation 4 seen in Section 3. The probability of a FP p in a Bloom filter is

$$\begin{aligned} p &= \Theta\left(\left[1 - \left(1 - \frac{1}{m}\right)^{kn}\right]^k\right) \\ &\approx \Theta\left(\left(1 - e^{-kn/m}\right)^k\right) \end{aligned}$$

where m is the number of bits, n is the number of inserted items and k is the number of hash functions.

Similarly, let m_i and k_i be the parameters of Bloom Filter i in a Bloom Vector of N Bloom Filters, and let n_i be the number of elements that the ith bloom filter contains. The expected FP rate is:

$$\text{FPR} = \prod_{i=0}^{N-1}\left[1 - \left(1 - \frac{1}{m_i}\right)^{k_i n_i}\right]^{k_i} \tag{13}$$

If a Bloom Vector is composed of Bloom Filters having the same parameters, we can simplify Equation 13. On average, if we call n_{tot} the total number of bits set to 1 in the Bloom Vector, each Bloom Filter contains $n = n_{\text{tot}}/N$ elements. We can

now write an equation for the average expected FP rate:

$$\mathrm{FPR} = \left[1 - \left(1 - \frac{1}{m}\right)^{kn_{\mathrm{tot}}/N}\right]^{Nk} \tag{14}$$

6.4 Complexity

Let us recall the complexity of a single Bloom Filter. The space occupied is $\Theta(m)$, while the time of the ADD function is $\Theta(k)$, and $O(k)$ for the LOOKUP function.

The complexity of the ENCODE(Π_m, S) function is $O(|S|)$, and the complexity of the DECODE(Π, V) function is $O(|V|)$.

In a Bloom Vector, we have a vector of N items, one for each $e \in E$, each of which contains a standard Bloom Filter. Let us call m the size of the biggest Bloom Filter. The space required is $O(mN)$. This is not a tight bound because different Bloom Filters in the Bloom Vector can have different sizes.

The ADD(l, e) operation needs to compute $V \leftarrow$ ENCODE(Π, e), and also to perform the ADD operation to each Bloom Filter corresponding to the indexes returned by ENCODE. Therefore, it takes $\Theta(N)$ space and $O(|e|) + O(|V|) \cdot \Theta(k) = O(|e| + |V|k)$ time, since the time for the bitwise OR operation is negligible.

A LOOKUP(l) operation needs to try a LOOKUP on each Bloom Filter. Therefore, it needs to set up a bitset V with all the bits representing the Bloom Filters in which the LOOKUP returns a positive set to 1; finally, it needs to return the value of DECODE(Π, V). The space taken is therefore $\Theta(N) + O(N) \cdot \Theta(k) = O(Nk)$. The time taken is $\Theta(N) \cdot \Theta(k) + O(|V|) = O(Nk + |V|)$.

6.5 Set operations

Similarly to Bloom filters, we can implement the set operations union and intersection between two Bloom Vectors. Let (G_1, Π_1) and (G_2, Π_2) be two Bloom Vectors. Let us define the Bloom Vector resulting from the operation as (G_r, Π_r). The operations are possible if and only if, for each $\Pi_1[i]$, $\Pi_2[j]$ for which $\Pi_1[i] = \Pi_2[j]$, $G_1[\Pi_1[i]]$ and $G_2[\Pi_2[j]]$ have the same size and hash functions.

Let us assume that there are some $\Pi_1[i]$ (or $\Pi_2[j]$) for which no $\Pi_2[j]$ (or $\Pi_1[i]$) is equal, that is, one item is represented in only one of the two Bloom Vectors. We just add each unique element to Π_r and its corresponding Bloom Filter to G_r if we are computing the union, or ignore it if we are computing the intersection.

In the pseudocodes that we show (Algorithm 10, 11), we assume that the two Bloom Vectors are compatible.

Algorithm 10 BV union

```
1: procedure UNION((G_1, Π_1, Π_1m), (G_2, Π_2, Π_2m))
2:     Π_∪ = Π_1 ∪ Π_2
3:     Π_∪m = Π_∪.zipWithIndex.toMap
4:     Let G_∪ be a new array of Bloom Filters of size |Π_∪|
5:     for i ← [0, |G_1| − 1] do
6:         j ← Π_∪m[Π_1[i]]
7:         G_∪[j] ← G_1[i]
8:     end for
9:     for i ← [0, |G_2| − 1] do
10:        j ← Π_∪m[Π_2[i]]
11:        G_∪[j] ← UNION(G_∪[j], G_2[i])
12:    end for
13:    return (G_∪, Π_∪, Π_∪m)
14: end procedure
```

Algorithm 11 BV intersection

```
1: procedure INTERSECTION((G_1, Π_1, Π_1m), (G_2, Π_2, Π_2m))
2:     Π_∩ = Π_1 ∩ Π_2
3:     Π_∩m = Π_∩.zipWithIndex.toMap
4:     Let G_∩ be a new array of Bloom Filters of size |Π_∩|
5:     for i ← [0, |Π_∩| − 1] do
6:         j ← Π_1[Π_∩[i]]
7:         k ← Π_2[Π_∩[i]]
8:         G_∩[i] ← G_1[j] ∩ G_2[k]
9:     end for
10:    return (G_∩, Π_∩, Π_∩m)
11: end procedure
```

6.6 Optimised Bloom vector

As we have seen in Section 3, if we know a priori the number of items to be inserted in a Bloom Filter, we can choose its parameters so that its probability of FP will

be around a chosen value. Let us assume that we know a priori the distribution of the values to be inserted in each of the Bloom Filters of a Bloom Vector. We can optimise each of its Bloom Filter, greatly reducing the memory overhead. We call this data structure Optimised Bloom Vector.

The Optimised Bloom Vector has the same time bounds of the Bloom Vector, but since its Bloom Filters can have different sizes, and thus the ones corresponding to items having a few labels can have much smaller sizes than the others, if there are many of such items, we can expect the Optimised Bloom Vector to require much less space by FP rate than the Bloom Vector. We will see this in practice in Section 9.

7 Bloom Matrix

Let us now introduce the second data structure: the Bloom Matrix. It works similarly to a Bloom Filter, with the difference that each bit in the bitset is replaced by another bitset of a fixed length, hence the name Bloom Matrix. We will see later that if we create a Bloom Matrix and a Bloom Vector with the same parameters, the Bloom Matrix is the transposed matrix of the matrix obtained creating a Bloom Vector.

7.1 Definition

We define a Bloom matrix as a quadruplet $(\mathbf{G}, \Pi, \Pi_m, H)$, where $\mathbf{G}$ is a binary matrix of size $m \times N$, Π and Π_m represent an ordering on the set E as previously defined, and $H = \{h_1(x), ..., h_k(x)\}$ is a set of hash functions, each having image $[0, m-1]$. If using MurmurHash, we can replace H with a number of hash functions k, and use as seeds for MurmurHash the range $[1, k]$. In the rest of the thesis we replace H with k.

7.2 Operations

Let us now take a look at the operations on a Bloom Matrix. These operations are the same as the ones on Bloom Vector, however, obviously, the implementation will be different.

Before defining such operations, we need to define another operation that we use in the actual implementation of our data structures, namely GetNeighbourhood.

7.2.1 GetNeighbourhood

GetNeighbourhood is a function to speed up operations in our data structures. It takes as parameters a label l, or a set of labels L, k representing the number of hash functions, and m representing the number of bits of a Bloom (Multi)Filter. The function returns a set of integers which represent the unique hashes of l or L with the k hash functions having image $[0, m-1]$. Let us show the implementations for both versions in Algorithm 12:

Algorithm 12 GetNeighbourhood

```
1: procedure GetNeighbourhood(l, k, m)
2:     Let I be a new set
3:     for i ← [1, k] do
4:         I.Add(MurmurHash(l, i))
5:     end for
6:     return I
7: end procedure
8:
9: procedure GetNeighbourhood(L, k, m)
10:    Let I be a new set
11:    for l ← L do
12:        for i ← [1, k] do
13:            I.Add(MurmurHash(l, i))
14:        end for
15:    end for
16:    return I
17: end procedure
```

7.2.2 Add

G is initialised with all its bits set to 0. Whenever we wish to add a label l to a Bloom Matrix F, we add the value returned by $\textsc{Encode}(\Pi_m, f(l))$ to the rows in its bit matrix **G** having the indexes equal to the hash neighbourhood of l, using the bitwise OR operator (Algorithm 13):

$$F.\,\textsc{Add}(l) := \mathbf{G}[h_i(l), \_] \leftarrow \mathbf{G}[h_i(l), \_] \vee \textsc{Encode}(\Pi_m, f(l))$$
$$\text{for } 1 \leq i \leq k$$

Algorithm 13 BM Add

```
1: procedure ADD(l, e)
2:     V ← ENCODE(Π_m, e)
3:     I ← GETNEIGHBOURHOOD(l, k, m)
4:     for i ← I do
5:         G[i, _] ← G[i, _] ∨ V
6:     end for
7: end procedure
```

7.2.3 Lookup

Whenever we wish to find out which subset of $\mathcal{P}(E)$ is labelled with l, we use the DECODE function on the bitset resulting from the bitwise AND operation on the rows in $\mathbf{G}$ having the indexes equal to the hash neighbourhood of l (Algorithm 14):

$$F.\,\textsc{Lookup}(l) := \textsc{Decode}\left(\Pi, \bigwedge_{1 \le i \le k} \mathbf{G}[h_i(l), \_]\right)$$

Algorithm 14 BM Lookup

```
1: procedure LOOKUP(l)
2:     I ← GETNEIGHBOURHOOD(l, k, m)
3:     Let V be an empty bitset
4:     for i ← I do
5:         V ← V ∧ G[i, _]
6:     end for
7:     return DECODE(Π, V)
8: end procedure
```

Multiple labels lookup Similarly to the Bloom Vector, we can lookup for multiple labels by computing the hash neighbourhood of all the labels, the rest of the lookup algorithm is identical to the algorithm for looking up a single label.

7.2.4 Loading data from a file

We can implement a function for loading data directly from a file also for the Bloom Matrix. The difference from loading it for a Bloom Vector is that, since we need to

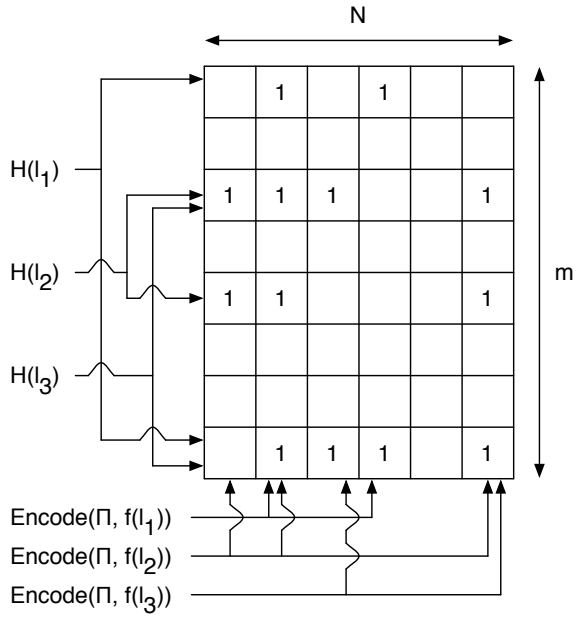


Figure 5: Example of a Bloom matrix, which is obtained from Example 7.1. The 1 in cell $\mathbf{G}[2, 5]$ is a collision.

know the ordering Π before creating the matrix, we need to do two scans of the file. The first scan reads only the first value in each line and creates Π, the second scan populates the matrix $\mathbf{G}$ according to Π and H.

Example 7.1. Let us assume that we have $E = \{e_1, ..., e_6\}$ with an ordering Π that follows the same order and $L = \{l_1, l_2, l_3\}$, and that we wish to represent a function $f : L \to \mathcal{P}(E)$ such that $f(l_1) = \{e_2, e_4\}$, $f(l_2) = \{e_1, e_2, e_6\}$ and $f(l_3) = \{e_3, e_6\}$. Let us assume that we create a Bloom matrix with $m = 8$ and $H = \{h_1(x), h_2(x)\}$ such that $H(l_1) = \{0, 7\}$, $H(l_2) = \{2, 4\}$ and $H(l_3) = \{2, 7\}$. We obtain a configuration as shown in Figure 5.

We can easily notice that LOOKUP(l_1) and LOOKUP(l_2) both yield correct results, but LOOKUP(l_3), which performs the bitwise AND operation on rows $\mathbf{G}[2, \_]$ and $\mathbf{G}[7, \_]$, returns $\{e_2, e_3, e_5\}$, which contains a FP (e_2).

7.3 Equivalence between Bloom Matrix and Bloom Vector

Let us assume that we encode a function f with a Bloom Matrix with size $m \times N$ and k hash functions, and a Bloom Vector of size N with each Bloom Filter of size m and same k hash functions. Furthermore, let us assume that we use the same ordering Π, therefore, the ENCODE and DECODE operations in both Bloom Matrix

and Bloom Vector on the same input yield the same result. We can show that the matrix constructed using the Bloom Vector is the transposed matrix constructed using the Bloom Matrix.

Suppose that we wish to add some label l to the Bloom Matrix and that we obtain a set of indexes $I = \{h_1(l), ..., h_k(l)\}$ from the hash functions, which represent rows in the Boolean matrix. Furthermore, suppose that we obtain a value from ENCODE($\Pi, f(l)$) in which the bits set to 1 are at the positions $J = \{j_1, ..., j_|J|\}$, which represent columns in the Boolean matrix. The ADD(l) function is going to set to 1 all the bits whose indexes are given by the Cartesian product $I \times J$.

Now, suppose that we wish to add the same label l to the Bloom Vector. The ADD(l) operation is going to be executed only on the rows $K = \{k_1, ..., k_|K|\}$ corresponding to the bits set to 1 in the value given by ENCODE($\Pi, f(l)$). The bits to be set to 1 in each of those rows are $M = \{h_1(l), ..., h_k(l)\}$, given by the hash functions. Therefore the bits that are going to be set to 1 are given by the Cartesian product $K \times M$, and since $K = J$ and $M = I$, it corresponds to the same positions in the transposed matrix.

Obviously, the FP rate is going to be the same for both structures. However, the Bloom Vector is also going to be slower than the Bloom Matrix, because during the LOOKUP operation it has to check each of the Bloom Filters. Therefore, using the Bloom Vector makes sense only if we exploit the possibility of having different sized Bloom Filters. We will discuss this in detail in Section 9.

7.4 False positive rate

By knowing all this, we can say that Equation 13 works also for the Bloom matrix, which we will see in Section 9 that it is also confirmed by experimental results.

7.5 Complexity

Let us assume that we have a Bloom matrix of size $m \times N$ and k hash functions. The ENCODE and DECODE functions need us to store all items of Π, plus we need to store the matrix and the hash functions. The exact space is therefore $2\Theta(N) + \Theta(mN) + \Theta(k) = \Theta(mN)$, since usually $k \ll mN$.

An ADD(l, e) operation on the Bloom Matrix needs to compute the neighbourhood of l and execute ENCODE(Π, e), then to update the matrix according to the results

obtained. The space taken by the neighbourhood of l is at most k, therefore $O(k)$; we have already seen that the space taken by the result of ENCODE is $\Theta(N)$, therefore, the total space taken is $\Theta(N)$, since usually $k \ll N$. The hash operations take $\Theta(k)$ time in total, while the ENCODE takes $O(|e| \log N)$ time. The insertion in the matrix takes $\Theta(N)$ time because there is only a bitwise OR operation on each row of the matrix for each value k, which is constant. The time of ADD is therefore $\Theta(k) + O(|e| \log N) + \Theta(k) = O(N \log |e|)$, since usually $k \ll N \log |e|$, and the time for the hash and bitwise OR operations are negligible.

A LOOKUP(l) operation needs to compute the neighbourhood of l and run DECODE(Π, V) on the V obtained by the bitwise AND operation on the rows having their index in the neighbourhood of l, which has a size of at most k, therefore the total space occupied is $\Theta(k) + O(N) = O(N)$, since usually $k \ll N$. The running time is $\Theta(k) + \Theta(k) = \Theta(k)$.

7.6 Set Operations

Similarly to Bloom Filters and Bloom Vectors, we apply the union and the intersection operations between Bloom matrices. Let $(G_1, \Pi_1, \Pi_{1m}, k_1)$ and $(G_2, \Pi_2, \Pi_{2m}, k_2)$ be two Bloom matrices. Let us define the Bloom matrix resulting from applying an operation as (G_r, Π_r, k). The operation is possible if and only if the two Bloom matrices have the same size and hash functions. Similarly to Bloom Vectors, If there are some $\Pi_1[i]$ (or $\Pi_2[j]$) for which no $\Pi_2[j]$ (or $\Pi_1[i]$) is equal, we just add it to Π_r and its corresponding column to G if we are computing the union, or ignore it if we are computing the intersection.

In the pseudocodes that we show (Algorithm 15, 16), we assume that the two Bloom Vectors are compatible.

7.7 Sparse Bloom Matrix (SBM)

If we use the standard Bloom matrix to encode a fixed function f, there is a chance that some rows will have unused bits at the end, wasting memory space.

If we know a priori the function f, we can construct a Bloom matrix with different row lengths that will occupy less memory space; we call it Sparse Bloom matrix. If we also compute the popularity of each item, we can sort the total ordering Π on the set E in decreasing order of number of labels assigned to each item. This

Algorithm 15 BM union

```
1: procedure UNION((G_1, Π_1, Π_1m, k_1), (G_2, Π_2, Π_2m, k_2))
2:     Π_∪ = Π_1 ∪ Π_2
3:     Π_∪m = Π_∪.zipWithIndex.toMap
4:     Let G_∪ be a new binary matrix having size |Π_∪| × |G_1[_, 1]|
5:     for i ← [0, |G_1[1, _]| − 1] do
6:         j ← Π_∪m[Π_1[i]]
7:         G_∪[_, j] ← G_1[_, i]
8:     end for
9:     for i ← [0, |G_2[1, _]| − 1] do
10:        j ← Π_∪m[Π_2[i]]
11:        G_∪[_, j] ← G_∪[_, j] ∨ G_2[_, i]
12:    end for
13:    return (G_∪, Π_∪, Π_∪m, k_1)
14: end procedure
```

Algorithm 16 BM intersection

```
1: procedure INTERSECTION((G_1, Π_1, Π_1m, k_1), (G_2, Π_2, Π_2m, k_2))
2:     Π_∩ = Π_1 ∩ Π_2
3:     Π_∩m = Π_∩.zipWithIndex.toMap
4:     Let G_∩ be a new binary matrix having size |Π_∩| × |G_1[_, 1]|
5:     for i ← [0, |G_∩[1, _]| − 1] do
6:         j ← Π_1m[Π_∩[i]]
7:         k ← Π_2m[Π_∩[i]]
8:         G_∩[_, i] ← G_1[_, j] ∧ G_2[_, k]
9:     end for
10:    return (G_∪, Π_∪, Π_∪m, k_1)
11: end procedure
```

maximises the probability of generating a high number of zeroes at the end when using ENCODE, further reducing memory usage.

More formally, let $C(e)$ be the number of labels assigned to an item e. We have to define the total ordering Π on S so that $\Pi(e_i) >_\Pi \Pi(e_j)$ if and only if $C(e_i) \leq C(e_j)$. Later in Section 9, we will see that this greatly reduces the space occupied in memory, especially in uniform distributions with a high density of bits set to 1.

Example 7.2. In Example 7.1, using different row sizes would make us spare 26

bits, but we can do better. Let us define the ordering Π such that the order of the items is $\{e_2, e_6, e_4, e_3, e_1, e_5\}$, this configuration, which is the Sparse Bloom Matrix, makes us spare 5 more bits, for a total of 31 bits.

7.8 Complexity

In the case of a Sparse Bloom Matrix, the ADD and LOOKUP operations are the same. However, the initialisation operation changes, because we need to compute the ordering Π. The speed of this operation depends on how the dataset is represented. If we represent it as an array of $|L|$ items, each item representing an $l \in L$ and containing a set of items of E, we need to scan the whole structure keeping a counter for each $e \in E$. The time would be therefore $\Theta(|L|N)$. If we represent it as an array of N items, each item representing an $e \in E$ and containing a set of labels of L, and if each set has a precomputed size, the time would be $\Theta(N)$.

8 Implementation

In this section, we discuss our implementation of our Bloom Multifilters, and also some other functions that we use for testing purposes. We implemented everything in Scala.

8.1 Hashing

As we already discussed in Section 2, we used MurmurHash for two reasons: because Scala has an API, and it can easily emulate the use of different hash functions.

We wrote two functions to compute a hash neighbourhood: one that computes it for one string, and one that computes it for multiple strings, given k and m. For MurmurHash we use consecutive integers from 1 to k as seeds. Since MurmurHash gives a 32-bit integer as a result, we compute the modulo operation by m on the hash to obtain our final result. We wrapped the hashing functions in an object called "Hashing".

```
object Hashing {
    def getIndexes(l: String, k: Int, m: Int): Set[Int] =
        (1 to k).map(i => (stringHash(l, i) & 0x7FFFFFFF) % m).toSet
```

```
    def getIndexes(L: Set[String], k: Int, m: Int): Set[Int] =
        L.flatMap(l => (1 to k).map(i => (stringHash(l, i) & 0x7FFFFFFF) % m))
}
```

We can notice a bitwise and operation with the value 0x7FFFFFFF. This is to force Scala into using positive integers.

8.2 Auxiliary Functions

For implementing Bloom Multifilters more easily, we implemented an object with auxiliary functions that we frequently used throughout the implementation. Let us show the code and discuss the functions one by one.

```
object BFAux {
    def optimalSize(n: Int, p: Double): Int =
        round(n.toDouble * -log(p) / (log(2.0)*log(2.0))).toInt

    def optimalHashN(p: Double): Int =
        max(round(-log(p)/log(2.0)).toInt, 1)

    def encode(Em: Map[String, Int], e: List[String]): BitSet = {
        var v = new BitSet(Em.size)

        e.foreach(l => v += Em(l))

        return v
    }

    def decode(E: Array[String], v: BitSet): List[String] =
        v.map((i: Int) => E(i)).toList

    def lookupRow(r: BitSet, nbh: Set[Int]): Boolean = {
        for (i <- nbh)
            if (!r.contains(i))
                return false

        return true
    }

    def lookupRow(r: Array[Byte], nbh: Set[Int]): Boolean = {
        for (i <- nbh)
            if (r(i) == 0)
                return false

        return true
    }
}
```

The first function computes the optimal size of a Bloom Filter, given the number

of elements that it will contain and the expected FP rate. The second computes its optimal number of hash functions, based on the expected FP rate. Both are based on Equations 6a and 6b that we discussed in Section 3:

$$m = -n\frac{\ln p}{\ln^2 2} \tag{6a}$$

$$k = -\log_2 p \tag{6b}$$

Unfortunately we cannot use these functions inside constructors because it is not allowed by Scala, but they still are useful in the rest of our implementation.

ENCODE and DECODE are implemented as in the pseudocodes but obviously adapted for Scala.

We also implemented a function called `lookupRow` in two versions, one for bitsets and one for arrays of bytes. Given a bitset and a set of integers, the function returns true if all integers are contained in such bitset. The second one returns true if all values with the indexes contained in the set of integers are different than 0.

We implemented these functions for two reasons. Firstly, sometimes we need to use the results of these operations inside conditions. Secondly, we can manually set a short circuit property by breaking the cycle, something that we can do only with a `return` statement inside a function, as Scala does not have the `break` statement.

8.3 Bloom Filter

To show the different values of FP rate obtained by the variation of m and k in Section 3, we implemented our Scala version of a Bloom Filter.

```
class BloomFilter(m: Int, k: Int) {
    import Hashing._
    import BFAux._

    private val F = new BitSet(m)

    def this(L: Set[String], m: Int, k: Int) {
        this(m, k)

        val I = getIndexes(L, k, m)
        F ++= I
    }

    def this(L: Set[String], p: Double) {
        this(L, round(L.size.toDouble * -log(p) / (log(2.0)*log(2.0))).toInt,
             round(-log(p)/log(2.0)).toInt)
```

```
    }
```

We can notice that we implemented three different constructors. The first is for creating an empty Bloom Filter specifying the parameters m and k. The second is for creating a Bloom Filter starting from a set by manually specifying the parameters. The last one is for creating a Bloom Filter from a set by specifying the expected FP rate, from which the parameters are chosen. Let us now see the other functions:

```
    def add(l: String): this.type = {
        val I = getIndexes(l, k, m)
        F ++= I

        return this
    }

    def lookup(l: String): Boolean = {
        if (F.size == 0)
            return false

        val I = getIndexes(l, k, m)

        return lookupRow(F, I)
    }
}
```

Both functions are trivial and are implemented almost exactly in the same way than in the pseudocodes.

8.4 Bloom Multifilters

We implemented a trait for our Bloom Multifilters so that we could have an interface to make them compatible.

```
trait MultiFilter {
    def add(l: String, e: List[String]): this.type
    def lookup(l: String): List[String]
    def lookup(L: Set[String]): List[String]
    def computeSpace(): Long
}
```

We can see that the trait specifies the ADD and LOOKUP functions. The other function, called `computeSpace`, computes the space occupied by the Bloom Multifilter, which is useful to evaluate the performance of our Bloom Multifilters.

8.5 Bloom Vector

We implement the Bloom Vector letting it directly take care of each row without using the Bloom Filter as a subclass. This approach is much faster and it also allows a greater control over the data. We implemented also multithreading on our Bloom Multifilters.

```
class BloomVector(private var E: Array[String], m: Int, private var k: Int)
      extends MultiFilter {
    import Hashing._
    import BFAux._

    private var M = if (E.size > 0) Array.fill(E.size){m}
                    else Array[Int]()
    private var F = if (M.size > 0) Array.fill(E.size){new BitSet(m)}
                    else Array[BitSet]()
    private var Em = if (E.size > 0) E.zipWithIndex.toMap
                     else Map[String, Int]()
    private val numth = Runtime.getRuntime.availableProcessors

    [...]
}
```

In the above class, `E` is an array which represents the ordering Π, while `Em` is a map which represents the inverted index of the ordering. `F` represents the vector of Bloom Filters, `M` contains the size of each Bloom Filter, and `numth` represents the number of threads. Normally it is set on the number of available processors. However, in this thesis we also tested the data structure with only one thread, to measure the difference in performance between a standard and multithreaded implementation.

We can see that we did not implement an array for differentiating the number of hash functions for each Bloom Filter. This is because the optimal k computed on the expected FP rate p is independent of the size m, therefore it is the same for all rows.

Let us now take a look at the methods of the class.

8.5.1 Constructors

We implemented a number of constructors, each useful for a different scenario.

The first two constructors are useful for loading data from a variable which represents the set. Using the first, a user can manually specify m and k, while using the second they can specify an expected FP rate p. We implemented both with

multithreading.

```
def this(data: Array[(String, Set[String])], m: Int, k: Int) {
    this(data.map(_._1), m, k)

    val threads = (0 until numth).toArray.map(th =>
        Future {
            for (i <- th until M.size by numth)
                F(i) ++= getIndexes(data(i)._2, k, M(i))
        }
    )

    threads.foreach(th => Await.result(th, Duration.Inf))

    Em = E.zipWithIndex.toMap
}
```

```
def this(data: Array[(String, Set[String])], p: Double) {
    this(data.map(_._1), 0, round(-log(p)/log(2.0)).toInt)

    val threads = (0 until numth).toArray.map(th =>
        Future {
            for (i <- th until M.size by numth) {
                M(i) = optimalSize(data(i)._2.size, p)
                F(i) = new BitSet(M(i)) ++ getIndexes(data(i)._2, k, M(i))
            }
        }
    )

    threads.foreach(th => Await.result(th, Duration.Inf))

    Em = E.zipWithIndex.toMap
}
```

As we can see, both constructors preallocate the Bloom Vector, so that each thread works on its assigned rows without the need for prepending or appending operations. Therefore, there is no need for synchronisation.

The last two constructors are useful for loading data from a file, and these too have different implementations. One in which a user can specify the parameters, and one in which they can choose the expected FP rate, and let the function specify them based on that.

```
def this(filename: String, m: Int, k: Int) {
    this(Array[String](), m, k)

    for (line <- Source.fromFile(filename).getLines) {
        val values = line.split(',')

        E = values(0) +: E
```

```
        F = (new BitSet(m) ++ getIndexes(values.drop(1).toSet, k, m)) +: F
        M +:= m
    }

    Em = E.zipWithIndex.toMap
}
```

```
def this(filename: String, p: Double) {
    this(Array[String](), 0, round(-log(p)/log(2.0)).toInt)

    for (line <- Source.fromFile(filename).getLines) {
        val values = line.split(',')
        val m = optimalSize(values.size - 1, p)

        E = values(0) +: E
        M = m +: M
        F = (new BitSet(m) ++ getIndexes(values.drop(1).toSet, k, m)) +: F
    }

    Em = E.zipWithIndex.toMap
}
```

8.5.2 Main operations

Add The add operation is very easy to adapt to a multithreaded implementation in Scala. Since we know a priori the rows in which we need to add the new element, we can partition the operation so that we do not need any synchronisation:

```
def add(l: String, e: List[String]): this.type = {
    val v = encode(Em, e).toArray

    val threads = (0 until min(numth, v.size)).toArray.map(th =>
        Future {
            for (i <- th until v.size by numth)
                F(v(i)) ++= getIndexes(l, k, M(i))
        }
    )

    threads.foreach(th => Await.result(th, Duration.Inf))

    return this
}
```

Lookup The lookup operation is easy to adapt for a multithreaded implementation, too. However, in this case, we need to put the result in a bitset. Since we do not know which bits we end up updating in each thread, we need to synchronise the

operations:

```
def lookup(l: String): List[String] = {
    var v = new BitSet(E.size)

    val threads = (0 until min(numth, E.size)).toArray.map(th =>
        Future {
            var vLocal = new BitSet(E.size)

            for (j <- th until E.size by numth) {
                if (M(j) > 0 && lookupRow(F(j), getIndexes(l, k, M(j))))
                    vLocal += j
            }

            this.synchronized {
                v |= vLocal
            }
        }
    )

    threads.foreach(th => Await.result(th, Duration.Inf))

    return decode(E, v)
}
```

The algorithm for multiple label lookup is almost identical to the algorithm for single label lookup. The only difference is that, instead of having the parameter `l` as `String`, we have a parameter `L` as `Set[String]`. Since `getIndexes` is overloaded for both versions, we do not need to change anything else.

8.5.3 Set operations

We implemented the set operations so that the Bloom Vector updates itself, instead of returning a new Bloom Vector. This requires some adaptation for the conversion from pseudocode to Scala.

Union Instead of creating a new ordering `E` with the union of the `E` of each Bloom Vector, we can simply prepend the elements of the second to the first and eliminate the duplicates. This preserves the original ordering of the elements belonging to the first. Then we can enlarge the number of rows with empty bitsets and the array containing the sizes with zeroes, and execute the rest of the algorithm:

```
def union(that: BloomVector): this.type = {
    this.E = (this.E ++ that.E).distinct
```

```
    this.Em = this.E.zipWithIndex.toMap

    while (this.E.size > this.F.size) {
        this.F :+= new BitSet()
        this.M :+= 0
    }

    for (i <- 0 until that.E.size) {
        val j = this.Em(that.E(i))

        this.F(j) |= that.F(i)
        this.M(j) = that.M(i)
    }

    return this
}
```

Intersection The intersection function is almost identical to its pseudocode version. However, the result is saved in the first Bloom Vector at the end, instead of creating a new one:

```
def intersect(that: BloomVector): this.type = {
    val newE = this.E intersect that.E
    val newEm = newE.zipWithIndex.toMap

    val newF = Array.fill(newE.size){new BitSet()}
    val newM = Array.fill(newE.size){0}

    for (i <- 0 until newE.size) {
        val j = this.Em(this.E(i))
        val k = that.Em(that.E(i))

        newF(i) = this.F(j) &= that.F(k)
        newM(i) = this.M(j)
    }

    this.E = newE
    this.Em = newEm
    this.F = newF
    this.M = newM

    return this
}
```

8.6 Bloom Matrix

Similarly to the Bloom Vector implementation, we implemented the Bloom Matrix using an array of bitsets.

```
class BloomMatrix(private var E: Array[String], var m: Int, k: Int)
      extends MultiFilter {
    import Hashing._
    import BFAux._

    private var F = if (m > 0) Array.fill(m){new BitSet(E.size)}
                    else Array[BitSet]()
    private var cmp = false
    private var Em = E.zipWithIndex.toMap

    [...]
}
```

`E` and `Em` work same as in the Bloom Vector, `F` is the array of bitsets which represents the matrix, and `cmp` is a variable which represents whether the Bloom Matrix is sparse or not.

8.6.1 Constructors

We implemented the constructors for the Bloom Matrix for the same use cases described for the Bloom Vector:

```
def this(data: Array[(String, Set[String])], m: Int, k: Int, compressed: Boolean) {
    this(data.map(_._1), m, k)

    cmp = compressed

    if (compressed) {
        data.sortBy(- _._2.size)
        E = data.map(_._1)
        Em = E.zipWithIndex.toMap

        for (i <- 0 until data.size) {
            val I = getIndexes(data(i)._2, k, m)
            val j = Em(data(i)._1)

            I.foreach(F(_).add(j))
        }
    }
    else {
        Em = E.zipWithIndex.toMap

        for (i <- 0 until data.size) {
            val I = getIndexes(data(i)._2, k, m)
            I.foreach(F(_).add(i))
        }
    }
}
```

```
def this(data: Array[(String, Set[String])], p: Double, compressed: Boolean) {
    this(data.map(_._1), 0, max(round(-log(p)/log(2.0)).toInt, 1))

    val nl = data.map(_._2.size).sum / data.size

    m = optimalSize(nl, p)

    cmp = compressed

    if (compressed) {
        data.sortBy(- _._2.size)
        E = data.map(_._1)
        F = Array.fill(m){new BitSet(E.size)}
        Em = E.zipWithIndex.toMap

        for (i <- 0 until data.size) {
            val I = getIndexes(data(i)._2, k, m)
            val j = Em(data(i)._1)

            I.foreach(F(_).add(j))
        }
    }
    else {
        F = Array.fill(m){new BitSet(E.size)}
        Em = E.zipWithIndex.toMap

        for (i <- 0 until data.size) {
            val I = getIndexes(data(i)._2, k, m)
            I.foreach(F(_).add(i))
        }
    }
}
```

In the first constructor, a Bloom Matrix is created with parameters given by the user, so the constructor only needs to populate it. We did not implement multithreading because the only operation with a significant amount of work would also need to be synchronised, which makes multithreading expensive. If we are trying to create a sparse Bloom Matrix, we need to sort the rows of the data in descending order by row size, in order to increase the probability of zeroes at the end of the rows in the matrix. The rest of the algorithm is the same in both cases. The difference is that, if we are creating a Sparse Bloom Matrix we need to sort the data first, and then we need to use the indexing `Em` when adding the elements.

In the second constructor, we need to estimate m. Since in the Bloom Matrix we cannot use a different m for each row, we compute it on the average number of elements by row, which is simply the mean of the number of elements for each row. The rest of the algorithm is identical to the first constructor, except that here we

need to allocate the matrix after we compute m.

The constructors which load the data from a file work pretty much the same way as the first two constructors. To get the data we need to scan the file two times, firstly to get the number of elements for each row, and secondly to scan it again to populate the matrix, since we need to sort the data first:

```
def this(filename: String, m: Int, k: Int, compressed: Boolean) {
    this(Array[String](), 0, k)

    cmp = compressed

    val counts = Source.fromFile(filename).getLines.toList.map(_.split(',')).
                 map(l => (l(0), l.size - 1))

    if (compressed) {
        counts.sortBy(- _._2)

        E = counts.map(_._1).toArray
        F = Array.fill(m){new BitSet(E.size)}
        Em = E.zipWithIndex.toMap

        for (line <- Source.fromFile(filename).getLines) {
            val values = line.split(',')
            val I = getIndexes(values.drop(1).toSet, k, m)
            val i = Em(values(0))

            I.foreach(F(_).add(i))
        }
    }
    else {
        E = counts.map(_._1).toArray
        F = Array.fill(m){new BitSet(E.size)}
        Em = E.zipWithIndex.toMap
        var i = 0

        for (line <- Source.fromFile(filename).getLines) {
            val values = line.split(',')
            val I = getIndexes(values.drop(1).toSet, k, m)

            I.foreach(F(_).add(i))
            i += 1
        }
    }

    this.m = m
}
```

```
def this(filename: String, p: Double, compressed: Boolean) {
    this(Array[String](), 0, max(round(-log(p)/log(2.0)).toInt, 1))

    cmp = compressed
```

```
    val counts = Source.fromFile(filename).getLines.toList.map(_.split(',')).
                map(l => (l(0), l.size - 1))

    m = optimalSize(counts.map(_._2).sum / counts.size, p)


    if (compressed) {
        counts.sortBy(- _._2)

        E = counts.map(_._1).toArray
        F = Array.fill(m){new BitSet(E.size)}
        Em = E.zipWithIndex.toMap

        for (line <- Source.fromFile(filename).getLines) {
            val values = line.split(',')
            val I = getIndexes(values.drop(1).toSet, k, m)
            val i = Em(values(0))

            I.foreach(F(_).add(i))
        }
    }
    else {
        E = counts.map(_._1).toArray
        F = Array.fill(m){new BitSet(E.size)}
        Em = E.zipWithIndex.toMap
        var i = 0

        for (line <- Source.fromFile(filename).getLines) {
            val values = line.split(',')
            val I = getIndexes(values.drop(1).toSet, k, m)

            I.foreach(F(_).add(i))
            i += 1
        }
    }
}
```

8.6.2 Main operations

We chose not to implement multithreading for the main operations too, for the same reason as in the constructors, namely because the only operation with a significant amount of work would also need to be synchronised.

Add The add operation is straightforward to implement, and it is identical to the pseudocode:

```
def add(l: String, e: List[String]): this.type = {
```

```
    val I = getIndexes(l, k, m)
    val v = encode(Em, e)

    I.foreach(F(_) |= v)

    return this
}
```

Lookup Similarly, the lookup operation is also straightforward, but we could simplify the algorithm using map-reduce. Instead of creating an empty bitset and populating it, we can directly select the rows with `map` and use `reduce` to perform the intersection, then pass them directly to `decode`:

```
def lookup(l: String): List[String] = {
    val I = getIndexes(l, k, m)

    return decode(E, I.map(F(_)).reduce(_ & _))
}
```

We can overload the `lookup` function so that it can deal with multiple labels. We can achieve this by changing `l` as `String` to `L` as `Set[String]`.

8.6.3 Set operations

We implemented the set operations in the same fashion as in the Bloom Vector, namely the Bloom Matrix updates itself instead of returning a new Bloom Matrix.

Union Since Scala automatically resizes bitsets, and we want to implement the Bloom Matrix so that it updates itself, implementing the function is trivial:

```
def union(that: BloomMatrix): this.type = {
    this.E = (this.E ++ that.E).distinct
    this.Em = this.E.zipWithIndex.toMap

    for (i <- 0 until that.m)
        that.F(i).foreach(j => this.F(i) += this.Em(that.E(j)))

    return this
}
```

Intersection The intersection operation is very similar to the pseudocode. However, it obviously required adaptation, since we wish the Bloom Matrix to update

itself instead of returning a new one:

```
def intersect(that: BloomMatrix): this.type = {
    val newE = this.E intersect that.E
    val newEm = newE.zipWithIndex.toMap

    val newF = Array.fill(this.m){new BitSet()}

    for (i <- 0 until this.m) {
        this.F(i).foreach(j => if (that.F(i).contains(that.Em(this.E(j))))
                               newF(i).add(newEm(this.E(j))))
    }

    this.E = newE
    this.Em = newEm
    this.F = newF

    return this
}
```

8.7 Datasets generation

We implemented two functions to generate the datasets that we use for testing our Bloom Multifilters. One generates uniform distributions, and the other generates Zipf distributions. Both functions save the generated data into CSV files. Each row of the file contains the name of an item as the first value and the names of all the labels assigned to it following in the same row.

Uniform The algorithm that generates uniform distributions, given E, L, and a probability p, for each $e \in E$, for each $l \in L$, decides with a probability of p whether to assign such label l to e or not:

```
def genUniform(E: Array[String], L: List[String], p: Double, filename: String):
               Int = {
    val file = new PrintWriter(new File(filename))
    var counter = 0

    for (i <- 0 until E.size) {
        val draw = L.filter(_ => Random.nextDouble <= p)
        counter += draw.size
        file.write(E(i) + "," + draw.reduce(_ + "," + _) + "\n")
    }

    file.close()
```

```
    return counter
}
```

Zipf The algorithm that generates Zipf distributions, given E, L and a real number s, generates the first $|E|$ Zipf rank numbers with exponent value s and $N = |E|$, following the equation:

$$f(k; s, n) = \frac{1/k^s}{\sum_{i=1}^{N}(1/n^s)} \tag{16}$$

Then, for each $e_k \in E$, for each $l \in L$, the algorithm decides with a probability equal to the rank k whether to assign such label l to e or not:

```
def zipfProb(N: Int, s: Double): Array[Double] = {
    var nsum = (1 to N).map(n => 1.0 / pow(n.toDouble, s)).sum

    return (1 to N).map(k => 1.0 / pow(k.toDouble, s) / nsum).toArray
}

def genZipf(E: Array[String], L: List[String], s: Double, filename: String): Int = {
    val file = new PrintWriter(new File(filename))
    var counter = 0

    var P = zipfProb(E.size, s)

    for (i <- 0 until E.size) {
        val draw = L.filter(l => Random.nextDouble <= P(i))
        counter += draw.size

        if (draw.isEmpty)
            file.write(E(i) + "\n")
        else
            file.write(E(i) + "," + draw.reduce(_ + "," + _) + "\n")
    }

    file.close()

    return counter
}
```

Part III

Experiments, Conclusions and references

In this last part of the thesis, we discuss experimental results from tests on our Bloom Multifilters with randomly generated data.

9 Experiments

We experimented with the two Bloom Multifilters, Bloom Vector and Bloom Matrix, on two types of distributions: uniform distribution and Zipf distribution.

The experiments were conducted on a machine with a quad-core processor at 2.3 GHz with eight logical processors, 16 GB of RAM at 1.6 GHz, and a 512 GB Apple SSD.

9.1 Datasets

We used two randomly generated datasets in our experiments. In the datasets, we used consecutive numbers as names of sets and of labels, with $|E| = 500$ for both datasets, $|L| = 10000$ for the Uniform dataset and $|L| = 30000$ for the Zipf dataset. Since in a real application the name of the sets and the labels can be strings of any kind, we treated such numbers as strings.

The first dataset was generated using the function `genUniform` with $p = 0.5$. The second dataset was generated with the function `genZipf` with $s = 0.8$. Both functions are defined in Section 8

Name	Number of labels	File size
Uniform	~ 2500000	12.2 MB
Zipf	~ 30000	171 kB

Table 1: Sizes of data sets used in the tests.

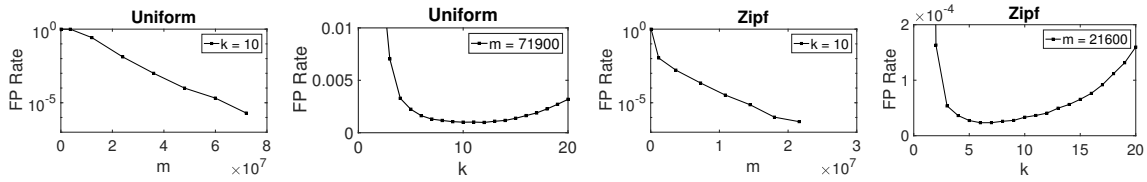


Figure 6: Bloom Matrix FP rate.

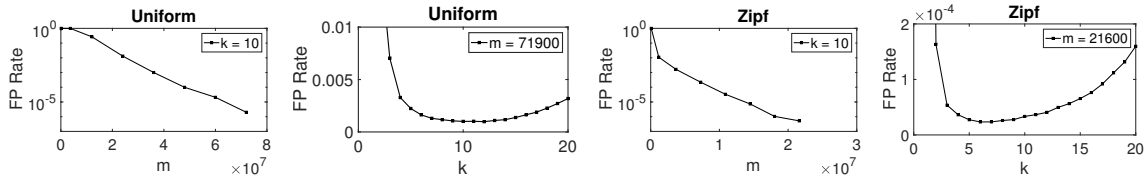


Figure 7: Bloom Vector FP rate.

9.2 Bloom Multifilters comparison

Let us now discuss the experimental results obtained from single threaded operations.

9.2.1 False positive rate

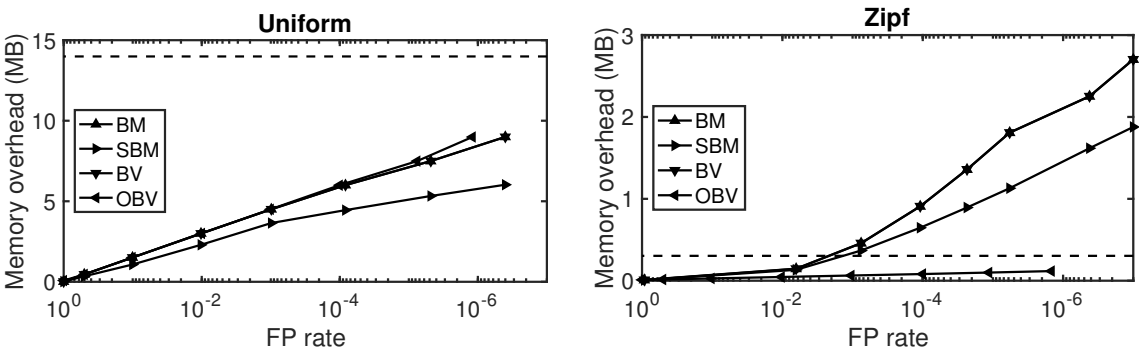
Let us discuss the FP rate of the Bloom Matrix and the Bloom Vector. As we can see in Figure 6 and Figure 7, the FP rate decreases as m increases, while with k decreases up to a value and starts increasing again, as expected from our theoretical analysis.

We notice also that the results are the same for Bloom Matrix and Bloom Vector, which confirms that, a Bloom Vector having all rows with the same size, and a Bloom Matrix having the same parameters, are equivalent.

9.2.2 Memory overhead by false positive rate

Let us now discuss the results in Figure 8. As we already discussed the Bloom Matrix and the Bloom Vector, in their basic variations, are equivalent. This implies that they occupy the same space by FP rate, both for uniform and Zipf distributions.

In Figure 8a, we notice that all versions of our Bloom Multifilters are well below the horizontal line, which represents memory usage if using conventional data structures to represent the data. The Optimised Bloom Vector occupies a similar space to its



(a) (b)

Figure 8: Memory usage by FP rate, using optimal m and k for each point. The horizontal dashed line represents memory usage if using conventional data structures to represent the data.

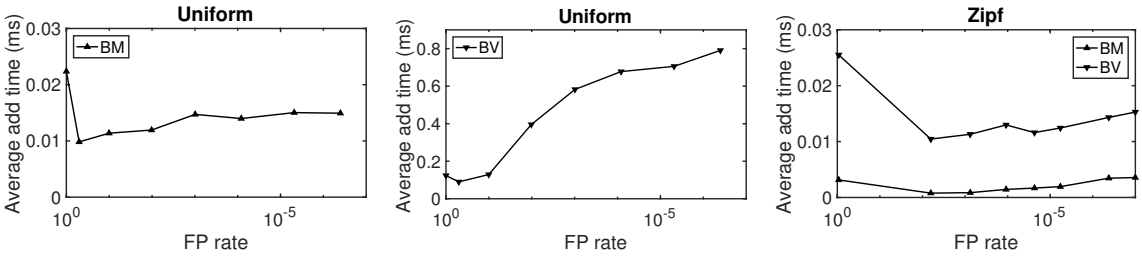
basic counterpart and the Bloom Matrix, while the Sparse Bloom Matrix occupies the least space. This is because, as we discussed before, the Sparse Bloom Matrix sets the ordering on the set E to maximise the number of zeroes at the end of each row. This makes it the most suitable Bloom Multifilter to represent uniformly distributed data if our goal is to spare as much space as possible.

In Figure 8b, on the other hand, we see that the standard Bloom Filter and Bloom Matrix perform poorly in terms of space. The Sparse Bloom Matrix performs better, but it is still well above the dashed line. The only variation that performs well in terms of memory in a Zipf distribution appears to be the Optimised Bloom Vector. The explanation is simple: the Optimised Bloom Vector uses different optimised sizes for each row, where each row represents the labels associated with a particular element in E. Therefore, with sparse rows of the distribution, it does not waste space as the other Bloom Multifilters do.

9.2.3 Operation times by false positive rate

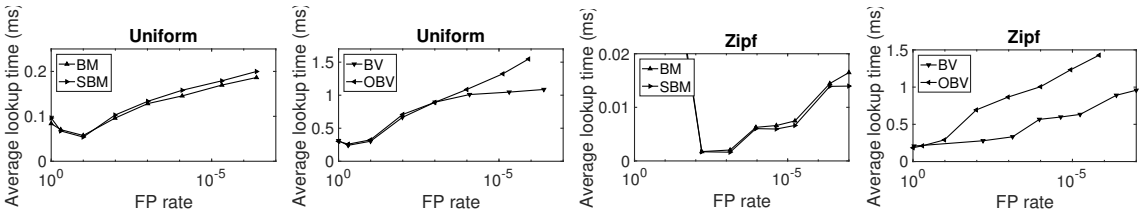
In Figure 9, we can see that the Bloom Matrix add time is near linear both in uniform and Zipf distributions as expected ($O(N \log |e|)$). The add operation on the Bloom Vector is linear too, but it takes more time than the Bloom Matrix. This is because the time depends on the size of the vector that it uses to store the result of ENCODE multiplied by k ($O(|e| + |V|k)$).

In Figure 10, we can see the lookup times. Both the Bloom Matrix and the Sparse



(a) (b) (c)

Figure 9: Add time by FP rate, using optimal m and k for each point.



(a) (b) (c) (d)

Figure 10: Lookup time by FP rate, using optimal m and k for each point.

Bloom Matrix have a similar lookup time, and both are faster than the Bloom Vector and the Optimised Bloom Vector in all situations. This is because the lookup time of the Bloom Matrix depends only on k ($\Theta(k)$). The Bloom Vector takes much more time and has a linear time too ($O(Nk + |V|)$). However, we notice a difference between its basic version and its optimised version. This might be because in the Optimised Bloom Vector each line is built with optimal parameters, and ends up having a higher concentration of bits, increasing the workload of the DECODE function.

9.2.4 Bloom Vector multithreading

Not surprisingly, we can notice in Figure 11 that the multithreaded version of the Bloom Vector is always better than a single-threaded implementation. We can also notice, however, that it is still slower than the Bloom Matrix. This is because the Bloom Matrix has $\Theta(k)$ complexity, while the Bloom Vector, even if multithreaded, still has complexity $O(Nk + |V|)$.

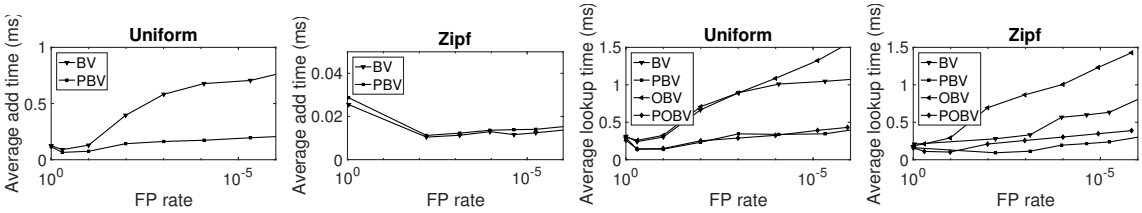


Figure 11: Add and lookup time differences between a single-threaded Bloom Vector and a multithreaded Bloom Vector.

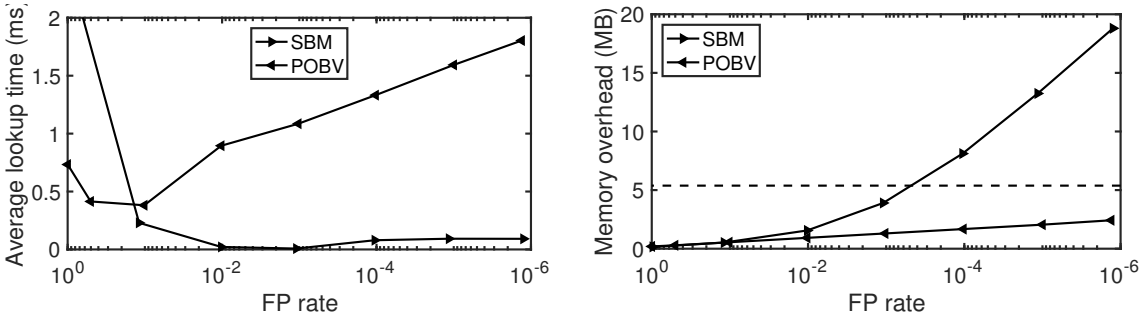


Figure 12: Average lookup time and memory overhead by FP rate on real data, using Sparse Bloom Matrix and Parallel Optimised Bloom Vector.

9.3 Test with real data

We also tested our Bloom Multifilters with real data, taken from the Datasets for single-label text categorisation [CC07]. Specifically, we used the 20ng-test-stemmed corpus, which is the 20 Newsgroups corpus test dataset, but with stemmed words.

9.3.1 Discussion

We tested the real data using only the best variations of our Bloom Multifilters, namely the Sparse Bloom Matrix and the Parallel Optimised Bloom Vector. As expected, the results are similar to the results obtained with our artificial Zipf dataset.

As we can see in Figure 12, both have a very quick lookup time, but the Sparse Bloom Matrix is much quicker than the Parallel Optimised Bloom Vector. The Parallel Optimised Bloom Vector always occupies less space than the data represented using conventional methods would, contrary to the Sparse Bloom Matrix. If using the Sparse Bloom Matrix, we can still spare space compared to the original data, with

a reasonable FP rate.

10 Conclusions

We discussed the Bloom Filter, some of its extensions and the theoretical knowledge necessary to understand how they work.

We also introduced two new data structures based on Bloom Filters, the Bloom Vector and the Bloom Matrix, and also optimised versions for each of them, namely the Optimised Bloom Vector and the Sparse Bloom Matrix.

The Bloom Matrix is the fastest data structure, therefore it is useful for applications in which the query time is critical. It is also useful in any situation in which we have to deal with uniformly distributed data since in that case, the Sparse Bloom Matrix is the structure having the least memory overhead.

When dealing with Zipf distributed data, however, all of the data structures except the Optimised Bloom Vector waste too much space above some FP rate threshold. Therefore, if in a particular application we need the FP rate to be very small, and if minimising memory overhead is more important than query speed, the Optimised Bloom Vector in such case is the logical choice.

10.1 Future work

As future work, it would be interesting to apply the compression algorithm or make scalable our Bloom Matrix and Bloom Vector, using the techniques that we have seen in the related work in Section 4 [Mit02, ABPH07].

References

ABPH07 Almeida, P. S., Baquero, C., Preguiça, N. and Hutchison, D., Scalable bloom filters. *Information Processing Letters*, 101,6(2007), pages 255–261.

BGK+08 Bose, P., Guo, H., Kranakis, E., Maheshwari, A., Morin, P., Morrison, J., Smid, M. and Tang, Y., On the false-positive rate of bloom filters. *Information Processing Letters*, 108,4(2008), pages 210–213.

Blo70 Bloom, B. H., Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, 13,7(1970), pages 422–426.

BM04 Broder, A. and Mitzenmacher, M., Network applications of bloom filters: A survey. *Internet Mathematics*, 1,4(2004), pages 485–509.

BMP+06 Bonomi, F., Mitzenmacher, M., Panigrahy, R., Singh, S. and Varghese, G. *An Improved Construction for Counting Bloom Filters*, pages 684–695. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

BP12 Brin, S. and Page, L., Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 56,18(2012), pages 3825–3833. The {WEB} we live in.

BW89 Baran, M. E. and Wu, F. F., Network reconfiguration in distribution systems for loss reduction and load balancing. *IEEE Transactions on Power Delivery*, 4,2(1989), pages 1401–1407.

CC07 Cardoso-Cachopo, A., Improving Methods for Single-label TextCategorization, PdD Thesis, Instituto Superior Tecnico, Universidade Tecnica de Lisboa, 2007.

CcFL04 Chang, F., chang Feng, W. and Li, K., Approximate caches for packet classification. *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, March 2004, pages 2196–2207.

CDG+08 Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R. E., Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26,2(2008), pages 4:1–4:26.

CKRT04 Chazelle, B., Kilian, J., Rubinfeld, R. and Tal, A., The bloomier filter: An efficient data structure for static support lookup tables. *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, Philadelphia, PA, USA, 2004, Society for Industrial and Applied Mathematics, pages 30–39.

Cyb89 Cybenko, G., Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7,2(1989), pages 279–301.

FCAB00 Fan, L., Cao, P., Almeida, J. and Broder, A. Z., Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8,3(2000), pages 281–293.

Gre82 Gremillion, L. L., Designing a bloom filter for differential file access. *Commun. ACM*, 25,9(1982), pages 600–604.

GWC+10 Guo, D., Wu, J., Chen, H., Yuan, Y. and Luo, X., The dynamic bloom filters. *IEEE Transactions on Knowledge and Data Engineering*, 22,1(2010), pages 120–133.

Hol03 Holzmann, G., *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

KT06 Kakoulis, K. G. and Tollis, I. G., Algorithms for the multiple label placement problem. *Computational Geometry*, 35,3(2006), pages 143–161.

KT13 Kakoulis, K. G. and Tollis, I. G., Labeling algorithms. In *Handbook of Graph Drawing and Visualization*, CRC Press, 2013, pages 489–515.

LR95 Li, Z. and Ross, K. A., Perf join: An alternative to two-way semijoin and bloomjoin. *Proceedings of the Fourth International Conference on Information and Knowledge Management*, CIKM '95, New York, NY, USA, 1995, ACM, pages 137–144.

Mit02 Mitzenmacher, M., Compressed bloom filters. *IEEE/ACM Transactions on Networking*, 10,5(2002), pages 604–612.

ML86 Mackert, L. F. and Lohman, G. M., R* optimizer validation and performance evaluation for distributed queries. *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, San Francisco, CA, USA, 1986, Morgan Kaufmann Publishers Inc., pages 149–159.

MNW98 Moffat, A., Neal, R. M. and Witten, I. H., Arithmetic coding revisited. *ACM Trans. Inf. Syst.*, 16,3(1998), pages 256–294.

MS15 Maggs, B. M. and Sitaraman, R. K., Algorithmic nuggets in content delivery. *SIGCOMM Comput. Commun. Rev.*, 45,3(2015), pages 52–66.

Mul83 Mullin, J. K., A second look at bloom filters. *Commun. ACM*, 26,8(1983), pages 570–571.

Mul90 Mullin, J. K., Optimal semijoins for distributed database systems. *IEEE Transactions on Software Engineering*, 16,5(1990), pages 558–560.

PBC00 Park, Y. W., Baek, K. H. and Chung, K. D., Reducing network traffic using two-layered cache servers for continuous media data on the internet. *Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International*, 2000, pages 389–394.

Rab89 Rabin, M. O., Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36,2(1989), pages 335–348.

RCP08 Rhea, S., Cox, R. and Pesterev, A., Fast, inexpensive content-addressed storage in foundation. *USENIX 2008 Annual Technical Conference*, ATC'08, Berkeley, CA, USA, 2008, USENIX Association, pages 143–156.

SB07 Swamidass, S. J. and Baldi, P., Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of Chemical Information and Modeling*, 47,3(2007), pages 952–964. PMID: 17444629.

SKH95 Shirazi, B. A., Kavi, K. M. and Hurson, A. R., editors, *Scheduling and Load Balancing in Parallel and Distributed Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1995.

XLR16 Xu, C., Liu, Q. and Rao, W. *BMF: An Indexing Structure to Support Multi-element Check*, pages 441–453. Springer International Publishing, Cham, 2016.

YCL16 Yin, S., Chen, D. and Le, J., Stnosql: Creating nosql database on the sensiblethings platform. *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, May 2016, pages 669–674.

YN13 Yamaguchi, F. and Nishi, H., Hardware-based hash functions for network applications. *2013 19th IEEE International Conference on Networks (ICON)*, Dec 2013, pages 1–6.

Appendix 1. Mathematics of Bloom Filters

FP rate derivation

Let us assume that we have a Bloom Filter of size m, with k hash functions with image $[0, m-1]$. Let us assume that the hash functions results follow a normal distribution and that we inserted n elements in the Bloom Filter. The probability that a single bit is not set to 1, after all the add operations, is:

$$\left(1 - \frac{1}{n}\right)^{kn}$$

The probability that such bit is set to 1 is therefore:

$$1 - \left(1 - \frac{1}{n}\right)^{kn}$$

We can now compute the probability that all bits are set to 1, which is also an upper bound on the FP rate:

$$\left[1 - \left(1 - \frac{1}{n}\right)^{kn}\right]^k \approx \left(1 - e^{-kn/m}\right)^k$$

Parameters derivation

To find the optimal k for a Bloom Filter, we compute the derivative of the probability that all bits are set to 1 and equate to 0. We find:

$$k = \frac{m}{n} \ln 2$$

By substituting it in the same equation we have:

$$p = \left(1 - e^{-\left(\frac{m}{n} \ln 2\right)\frac{n}{m}}\right)^{\frac{m}{n} \ln 2}$$

$$\ln p = -\frac{m}{n}(\ln 2)^2$$

$$m = -n\frac{\ln p}{\ln^2 2}$$